```
; -----------------------------------------------------------------------------
; Progetto : Realizzazione di un risolutore e di un generatore di problemi di
;            CSP binario.
; Autori   : Alberto Schena            (#60735)
;            Nicola Fraccaroli         (#60733)
;            Giovanni Dall'Oglio Risso (#60734)
; Corso    : Intelligenza Artificiale A
; Docente  : Alfonso Gerevini
; -----------------------------------------------------------------------------

; -----------------------------------------------------------------------------
; SPECIFICHE:
; -----------------------------------------------------------------------------

; ASSEGNAMENTO:
;   Lista di associazione dalla forma:
;     ((var1-symbol value1) (var2-symbol value2) ... (varN-symbol valueN))
;   in cui ogni coppia definisce il valore numerico di una variabile di un problema
;   di CSP.
; DOMINIO:
;   Costituito dalla coppia:
;     (var-symbol (value1 value2 ... valueN))
;   che realizza l'associazione tra una variabile e l'insieme dei suoi valori legali
;   per il problema. La lista dei valori non ammette duplicati dei suoi elementi.
; DEFINIZIONE DI VARIABILE:
;   Sintatticamente analoga al DOMINIO di una variabile ma rappresenta l'associazione
;   tra una variabile ed i soli valori del suo dominio che sono consistenti con tutti i
;   vincoli del problema. Per variabili fissate da un assegnamento la coppia assume la
;   forma ridotta (var-symbol (assigned-value)).
; LISTA DEI DOMINII:
;   Lista di associazione in cui ogni elemento è la definizione di un DOMINIO. Essa
;   assume pertanto la forma:
;   ((var1-symbol (value1 value2 ... valueN1))
;    (var2-symbol (value1 value2 ... valueN2))
;    ...
;    (varK-symbol (value1 value2 ... valueNK)))
; FILE DEI DOMINII:
;   File di testo organizzato per righe. Ciascuna riga assume la forma:
;     (var-symbol value1 value2 ... valueN)
;   in cui:
;   - var-symbol identifica una variabile del problema;
;   - value1..N definiscono il dominio di tale variabile.
;   I dominii possono contenere solamente valori interi. Ogni altro simbolo letto
;   da file viene automaticamente rimosso.
; VINCOLO (BINARIO):
;   Costituito dalla terna:
;     (rel-op var1-symbol var2-symbol)
;   in cui:
;   - rel-op è un symbol e può valere: =, /=, <, <=, >, oppure >=.
;   - var1-symbol e var2-symbol sono symbol che indicano rispettivamente le variabili
;     a sinistra ed a destra dell'operatore di confronto. Non sono ammessi
;     auto-vincoli, ovvero auto-anelli nel grafo dei vincoli.
; LISTA DEI VINCOLI:
;   Lista in cui ogni elemento è la definizione di un VINCOLO. Essa assume pertanto la
;   forma:
;     ((rel-op1 var11-symbol var12-symbol)
;      (rel-op2 var21-symbol var22-symbol)
;      ...
```

```
;      (rel-opJ varJ1-symbol varJ2-symbol))
; FILE DEI VINCOLI:
;   File di testo organizzato per righe. Ciascuna riga contiene la terna:
;     (var1-symbol rel-op var2-symbol)
;   ovvero l'equivalente in notazione infissa dell'operatore di confronto della terna
;   del vincolo binario. var1-symbol e var2-symbol devono essere variabili definite nel
;   file dei dominii. In caso di variabile non definita viene generato un errore.
;   Se var1-symbol e var2-symbol coincidono (auto-vincolo, ovvero auto-anello nel
;   grafo dei vincoli) viene generato un errore.
; ARCO ORIENTATO (del grafo dei vincoli) ed ARCO INVERSO:
;   Sintatticamente analogo ad un VINCOLO BINARIO. var1-symbol e var2-symbol
;   rappresentano rispettivamente le variabili iniziale e finale dell'arco orientato.
;   Ad un vincolo corrispondono due archi orientati, uno diretto ed uno inverso.
;   Segue una tabella di corrispondenze tra vincoli, archi diretti ed inversi:
;     VINCOLO     ARCO DIRETTO     ARCO INVERSO
;     (= x y)     (= x y)          (= y x)
;     (/= x y)    (/= x y)         (/= y x)
;     (< x y)     (< x y)          (> y x)
;     (<= x y)    (<= x y)         (>= y x)
;     (> x y)     (> x y)          (< y x)
;     (>= x y)    (>= x y)         (<= y x)

; ---------------------------------------------------------------------------------
; COSTANTI
; ---------------------------------------------------------------------------------

(defconstant ERR-01 "Riga ~a: Dominio vuoto per la variabile '~a'.")
(defconstant ERR-02 "Riga ~a: Operatore '~a' non valido.")
(defconstant ERR-03 "Riga ~a: Variabile '~a' non definita.")
(defconstant ERR-04 "Riga ~a: Auto-vincolo sulla variabile '~a'.")

; ---------------------------------------------------------------------------------
; MACRO
; ---------------------------------------------------------------------------------

; NOME:
;   get-min
; DESCRIZIONE:
;   Restituisce il valore minimo nella lista 'lst'.
; PARAMETRI:
;   lst (tipo: list)
;     La lista in cui cercare il valore minimo.
; RITORNO:
;   (tipo: number) Il valore dell'elemento minimo.
; NOTE:
;   `(loop for i in ,lst minimize i) impiega troppa memoria rispetto a questa
;   implementazione. Test eseguito con (time ...).
(defmacro get-min (lst) `(apply #'min ,lst))
; ---------------------------------------------------------------------------------

; NOME:
;   get-max
; DESCRIZIONE:
;   Restituisce il valore massimo nella lista 'lst'.
; PARAMETRI:
;   lst (tipo: list)
;     La lista in cui cercare il valore massimo.
; RITORNO:
```

```
;   (tipo: number) Il valore dell'elemento massimo.
; NOTE:
;   `(loop for i in ,lst maximize i) impiega troppa memoria rispetto a questa
;   implementazione. Test eseguito con (time ...).
(defmacro get-max (lst) `(apply #'max ,lst))
; -------------------------------------------------------------------------------

; NOME:
;   enqueue
; DESCRIZIONE:
;   Appende il valore di 'x' in coda alla lista 'queue'.
; PARAMETRI:
;   queue (tipo: list)
;     La lista a cui appendere l'elemento. Deve essere una variabile.
;   x (tipo: <qualsiasi>)
;     L'elemento da appendere.
; RITORNO:
;   (tipo: list) La lista modificata.
; SIDE-EFFECTS:
;   Modifica il contenuto di 'queue'.
(defmacro enqueue (queue x) `(setf ,queue (append ,queue ,x)))
; -------------------------------------------------------------------------------

; NOME:
;   dequeue
; DESCRIZIONE:
;   Preleva l'elemento in testa alla lista 'queue'.
; PARAMETRI:
;   queue (tipo: list)
;     La lista da cui prelevare l'elemento. Deve essere una variabile.
; RITORNO:
;   (tipo: list) Il primo elemento di 'queue'.
; SIDE-EFFECTS:
;   Modifica il contenuto di 'queue'.
(defmacro dequeue (queue) `(prog1 (car ,queue) (setf ,queue (cdr ,queue))))
; -------------------------------------------------------------------------------

; -------------------------------------------------------------------------------
; STRUTTURE
; -------------------------------------------------------------------------------

; NOME:
;   csp
; DESCRIZIONE:
;   Descrive il modello di un problema di CSP binario.
(defstruct csp
    ; Lista dei dominii delle variabili del problema.
    (domains nil)
    ; Lista di vincoli (binari) del problema.
    (constraints nil)
    ; Lista dei nomi delle variabili del problema (informazione ridondante mantenuta
    ; nella struttura per maggiore efficienza dell'algoritmo di risoluzione).
    (variables nil)
)
; -------------------------------------------------------------------------------

; -------------------------------------------------------------------------------
; FUNZIONI
```

```
; -------------------------------------------------------------------------------

; NOME:
;   deep-copy-csp
; DESCRIZIONE:
;   Restituisce un clone indipendente del modello di problema 'csp'.
;   Effettua una deep-copy dei campi. Da usare quando è necessario separare a livello di
;   memoria il clone dalla struttura originaria.
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello di problema da clonare.
; RITORNO:
;   (tipo: csp) Il clone del problema
; NOTE:
;   copy-csp effettua invece una shallow-copy dei campi della struttura dati.
;   I campi del clone puntano pertanto alla stessa memoria allocata per i corrispondenti
;   campi della struttura originaria.
(defun deep-copy-csp (csp)
    (make-csp
        :domains (copy-tree (csp-domains csp))
        :constraints (copy-tree (csp-constraints csp))
        :variables (copy-tree (csp-variables csp))
    )
)
; -------------------------------------------------------------------------------

; NOME:
;   csp-load
; DESCRIZIONE:
;   Carica il modello del problema di CSP dai file specificati per i dominii ed i
;   vincoli, rispettivamente.
; PARAMETRI:
;   dom-file-path (tipo: string)
;     Il percorso del FILE DEI DOMINII.
;   con-file-path (tipo: string)
;     Il percorso del FILE DEI VINCOLI.
; RITORNO:
;   (tipo: csp) Il modello caricato.
(defun csp-load (dom-file-path con-file-path)
    (let* ((dom-list (read-domains dom-file-path))
           (var-list (mapcar #'car dom-list))
           (con-list (read-constraints con-file-path var-list)))
        (make-csp
            :domains dom-list
            :constraints con-list
            :variables var-list
        )
    )
)
; -------------------------------------------------------------------------------

; NOME:
;   csp-solve
; DESCRIZIONE:
;   Risolve il problema 'csp' con backtracking e ne restituisce la prima soluzione
;   trovata (se esiste). Ritorna 'INFEASIBLE se il problema non ammette alcuna soluzione.
; PARAMETRI:
;   csp (tipo: csp)
```

```
;     Il modello di problema da risolvere.
;   &key use-mrv (tipo: boolean; default: T)
;     La flag che indica se utilizzare l'euristica MRV (Minimum Remaining Values) nella
;     selezione delle variabili.
;   &key use-mcv (tipo: boolean; default T)
;     La flag che indica se utilizzare l'euristica di grado MCV (Most Constrained
;     Variable) nella selezione delle variabili.
;   &key use-ac3 (tipo: boolean; default: T)
;     La flag che indica se utilizzare AC-3 (Arc Consistency) nella risoluzione.
;   &key verbose (tipo: boolean; default: T)
;     La flag che attiva la stampa a video di informazioni sulla risoluzione e dei
;     risultati.
; RITORNO:
;   Si veda RITORNO di csp-backtrackng.
(defun csp-solve (csp &key (use-mrv t) (use-mcv t) (use-ac3 t) (verbose t))
    (when verbose
        (princ (format nil "Risolutore CSP~%"))
        (princ (format nil "MRV: ~a~%" (if use-mrv "ON" "OFF")))
        (princ (format nil "MCV: ~a~%" (if use-mcv "ON" "OFF")))
        (princ (format nil "AC3: ~a~%" (if use-ac3 "ON" "OFF")))
    )
    (let ((result (csp-backtracking csp nil use-mrv use-mcv use-ac3)))
        (when verbose
            (if (symbolp result)
                (print result)
                (loop for x in result do
                    (princ (format nil "~a = ~a~%" (car x) (cadr x)))
                )
            )
        )
        result
    )
)
; -----------------------------------------------------------------------------

; NOME:
;   csp-backtracking
; DESCRIZIONE:
;   Esegue la ricerca in profondità di un assegnamento completo a partire da
;   'assignment' che sia soluzione del problema 'csp'.
; PARAMETRI:
;   csp (tipo: csp)
;     Il problema da risolvere.
;   assignment (tipo: list)
;     L'assegnamento corrente da analizzare.
;   use-mrv (tipo: boolean)
;     La flag che indica se utilizzare l'euristica MRV (Minimum Remaining Values).
;   use-mcv (tipo: boolean)
;     La flag che indica se utilizzare l'euristica di grado MCV (Most Constrained
;     Variable).
;   use-ac3 (tipo: boolean)
;     La flag che indica se utilizzare l'algoritmo AC-3 per mantenere l'arc-consistency.
; RITORNO:
;   (tipo: list oppure symbol)
;     L'assegnamento completo che costituisce la soluzione trovata per il problema.
;     Se il problema non ammette soluzione restituisce invece il symbol 'INFEASIBLE.
; ALGORITMO:
;   if IS-COMPLETE(csp, assignment) then return assignment
```

```
;   var = SELECT-UNASSIGNED-VARIABLE(csp, assignment)
;   for each val in SORT-VARIABLE-DOMAIN(csp, assignment, var) do
;       if VERIFY-NEW-ASSIGNMENT(csp, assignment, {var = val}) then
;           new-assignment = assignment + {var = val}
;           result = CSP-BACKTRACKING(csp, new-assignment)
;           if result != FAILURE then return result
;   return FAILURE
; NOTE:
;   Se abilitato, AC-3 viene eseguito sia in fase di pre-processing che in seguito
;   ad ogni nuovo assegnamento.
(defun csp-backtracking (csp assignment use-mrv use-mcv use-ac3)
    ; Arc-consistency in fase di pre-processing e ad ogni nuovo assegnamento.
    (when use-ac3 (setq csp (ac3 csp)))

    (if (is-complete csp assignment)
        assignment
        (do* ((var (select-unassigned-variable csp assignment
                   use-mrv use-mcv))
              (domain (get-variable-domain csp var))
              (new-csp (deep-copy-csp csp)))
             ((endp domain) 'INFEASIBLE)

             (let* ((val (pop domain))
                    (new-assignment (append assignment (list (list var val)))))

                 ; Restringe il dominio della variabile assegnata al solo valore
                 ; corrente. Si veda DEFINIZIONE DI VARIABILE.
                 ; NOTA: Il modello 'new-csp' è un clone locale del modello "padre",
                 ;       'csp' pertanto il backtracking consiste semplicemente
                 ;       nell'abbandono del modello "figlio".
                 (set-variable-domain new-csp var (list val))

                 ; DEBUG ---
                 ;(princ (format nil "DEBUG: ~a = ~a~%" var val))
                 ; DEBUG ---

                 (when (verify-assignment new-csp new-assignment)
                     (let* ((result (csp-backtracking new-csp new-assignment
                                     use-mrv use-mcv use-ac3)))
                         (unless (symbolp result)
                             (return result)
                         )
                     )
                 )
             )
        )
    )
)
; -----------------------------------------------------------------------------

; NOME:
;   ac3
; DESCRIZIONE:
;   Algoritmo di MAC (Maintaining Arc-Consistency). Restituisce un modello di problema
;   di CSP analogo a quello specificato ma con dominii ridotti in modo da mantenere
;   l'arc-consistency.
; PARAMETRI:
;   csp (tipo: csp)
```

```
;     Il modello di problema da analizzare.
; RITORNO:
;   (tipo: csp)
;     Il modello arc-consistent ottenuto.
; ALGORITMO:
;   csp2 = DEEP-COPY(csp)
;   queue = ORIENTED-ARCS(csp)
;   while NOT-EMPTY(queue) do
;       [Xi, Xj] = DEQUEUE(queue)
;       removed = false
;       for each x in DOMAIN[Xi] do
;           if nessun valore y in DOMAIN[Xj] soddisfa VINCLE[Xi, Xj](x, y) then
;               REMOVE(x, DOMAIN[Xi])
;               removed = true
;       if removed then
;           for each Xk in ADJACENT[Xi] do
;               ENQUEUE([Xk, Xi], queue)
;   return csp2
(defun ac3 (csp)
    (do ((csp2 (deep-copy-csp csp))
         (arcs (get-arcs csp))
         (queue (get-arcs csp)))
        ((endp queue) csp2)

        (let* ((xi-xj (dequeue queue))
               (removed nil)
               (xi (cadr xi-xj))
               (xj (caddr xi-xj))
               (xi-domain (get-variable-domain csp2 xi))
               (xj-domain (get-variable-domain csp2 xj)))
            (loop for x in xi-domain do
                (when (loop for y in xj-domain never
                    (verify-constraint (list (list xi x) (list xj y)) xi-xj))

                    ; DEBUG ---
                    ; (princ (format nil "Removing ~a from ~a~%" x xi))
                    ; DEBUG ---
                    (remove-variable-value csp2 xi x)
                    (setq removed t)
                )
            )
            (when removed
                (enqueue queue (get-incoming-arcs arcs xi))
            )
        )
    )
)
; -----------------------------------------------------------------------------

; NOME:
;   get-incoming-arcs
; DESCRIZIONE:
;   Restituisce la lista degli ARCHI ORIENTATI entranti nella variabile 'var'.
; PARAMETRI:
;   arcs (tipo: lista)
;     La lista di archi orientati in cui effettuare la ricerca.
;   var (tipo: symbol)
;     La variabile del problema per la quale effettuare la ricerca.
```

```
; RITORNO:
;   (tipo: list)
;     Sottolista di 'arcs' contenente i soli archi entranti nella variabile 'var'.
(defun get-incoming-arcs (arcs var)
    (loop for x in arcs
        when (eql (caddr x) var) collect x
    )
)
; -----------------------------------------------------------------------------

; NOME:
;   get-arcs
; DESCRIZIONE:
;   Restituisce la lista di tutti gli ARCHI ORIENTATI (diretti ed inversi) presenti
;   nel grafo dei vincoli del problema 'csp'.
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello di problema da analizzare.
; RITORNO:
;   (tipo: list)
;     La lista degli archi orientati del problema.
(defun get-arcs (csp)
    (let ((result nil))
        (dolist (x (csp-constraints csp) result)
            (setq result
                (append result
                    (list x)
                    (list (get-inverse-arc x))
                )
            )
        )
    )
)
; -----------------------------------------------------------------------------

; NOME:
;   get-inverse-arc
; DESCRIZIONE:
;   Restituisce l'ARCO INVERSO rispetto ad 'arc'.
; PARAMETRI:
;   arc (tipo: list)
;     L'arco da invertire.
; RITORNO:
;   (tipo: list)
;     L'arco inverso rispetto ad 'arc'.
(defun get-inverse-arc (arc)
    (list
        (case (car arc)
            ('= '=) ('/= '/=) ('< '>) ('> '<) ('<= '>=) ('>= '<=)
        )
        (caddr arc)
        (cadr arc)
    )
)
; -----------------------------------------------------------------------------

; NOME:
;   verify-assignment
```

```
; DESCRIZIONE:
;   Verifica la validità dell'ASSEGNEMENTO 'assignment' rispetto ai VINCOLI presenti nel
;   problema 'csp'.
; PARAMETRI:
;   csp (tipo: csp)
;     Il problema i cui vincoli devono essere analizzati.
;   assignment (tipo: list)
;     L'assegnamento da verificare.
; RITORNO:
;   (tipo: boolean)
;     Ritorna NIL se l'assegnamento viola almeno un vincolo del problema, altrimenti
;     ritorna 'assignment'.
(defun verify-assignment (csp assignment)
    ; DEBUG ---
    ;(princ (format nil "~a~%" assignment))
    ; DEBUG ---
    (dolist (x (csp-constraints csp) assignment)
        ; DEBUG ---
        ;(princ (format nil "~a" x))
        ; DEBUG ---
        (unless (verify-constraint assignment x)
            (return nil)
        )
    )
)
; -------------------------------------------------------------------------------------

; NOME:
;   verify-constraint
; DESCRIZIONE:
;   Verifica che il VINCOLO 'constraint' sia rispettato dall'ASSEGNAMENTO 'assignment'.
; PARAMETRI:
;   assignment (tipo: list)
;     L'assegnamento da utilizzare nella valutazione.
;   constraint (tipo: list)
;     Il vincolo da valutare.
; RITORNO:
;   (tipo: boolean)
;     Ritorna NIL se entrambe le variabili coinvolte da 'constraint' sono definite
;     in 'assignment' ed il vincolo è violato, altrimenti restituisce T.
;     Restituisce pertanto T in due casi distinti:
;       1. Almeno una delle variabili non è definita nell'assegnamento.
;       2. Entrambe le variabili sono definite ed il vincolo è rispettato.
(defun verify-constraint (assignment constraint)
    (let* ((arg1 (cadr (assoc (cadr constraint) assignment)))
           (arg2 (cadr (assoc (caddr constraint) assignment)))
           (vc (list (car constraint) arg1 arg2)))

        (if (and arg1 arg2)
            (eval vc)
            t
        )
    )
)
; -------------------------------------------------------------------------------------

; NOME:
;   select-unassigned-variable
```

```
; DESCRIZIONE:
;   Seleziona la prossima variabile da assegnare per la risoluzione del problema 'csp'
;   in base all'ASSEGNAMENTO 'assignment'.
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello di problema da analizzare.
;   assignment (tipo: list)
;     L'assegnamento corrente in base al quale determinare le variabili non assegnate.
;   use-mrv (tipo: boolean)
;     La flag che indica se utilizzare l'euristica MRV (Minimum Remaining Values) nella
;     selezione della variabile.
;   use-mcv (tipo: boolean)
;     La flag che indica se utilizzare l'euristica di grado MCV (Most Constrained
;     Variable) nella selezione della variabile.
; RITORNO:
;   (tipo: symbol)
;     La variabile candidata per il prossimo assegnamento.
(defun select-unassigned-variable (csp assignment use-mrv use-mcv)
    ; unvardefs: DEFINIZIONE DELLE VARIABILI non assegnate.
    (let ((unvardefs (remove-if #'(lambda (x) (assoc (car x) assignment))
            (csp-domains csp))))

        ; --- Euristica MRV:
        ; Mantiene in 'unvardefs' solo le DEFINIZIONI DI VARIABILI aventi dominio
        ; con cardinalità minima.
        (when (and use-mrv (> (length unvardefs) 1))
            (setq unvardefs (get-mrv-vardefs unvardefs))
        )

        ; --- Euristica MCV:
        ; Mantiene in 'unvardefs' solo le DEFINIZIONI DELLE VARIABILI coinvolte nel
        ; maggior numero di vincoli con le altre variabili non assegnate.
        (when (and use-mcv (> (length unvardefs) 1))
            (setq unvardefs (get-mcv-vardefs csp unvardefs))
        )

        ; Per eliminare (ulteriori) pareggi seleziona la prima delle definizioni
        ; ottenute.
        (caar unvardefs)
    )
)
; -----------------------------------------------------------------------------

; NOME:
;   get-mrv-vardefs
; DESCRIZIONE:
;   Implementa l'euristica MRV (Minimum Remaining Values).
;   L'algoritmo confronta le cardinalità degli insiemi di definizione di ciascuna
;   variabile definita in 'unvardefs' e restituisce le sole DEFINIZIONI DI VARIABILI
;   aventi cardinalità minima (minor numero di valori assegnabili rimanenti).
;   Per utilizzare correttamente questa funzione occorre fornire le definizioni
;   delle sole variabili non ancora assegnate.
; PARAMETRI:
;   vardefs (tipo: list)
;     La lista delle definizioni delle variabili da analizzare.
; RITORNO:
;   (tipo: list)
;     Restituisce la lista di DEFINIZIONI DELLE VARIABILI aventi dominio con cardinalità
```

```
;     minima.
; NOTE:
;   Il ritono è una lista: in caso di tie (pareggio) si dovrà filtrare ulteriormente.
(defun get-mrv-vardefs (unvardefs)
    (let ((min-card (get-min (mapcar #'(lambda (x)
            (length (cadr x))) unvardefs))))

        (remove-if
            #'(lambda (x) (> (length (cadr x)) min-card))
            unvardefs
        )
    )
)
; ------------------------------------------------------------------------------

; NOME:
;   get-mcv-vardefs
; DESCRIZIONE:
;   Implementa l'euristica di grado MCV (Most Constrained Variable) per la selezione
;   della variabile per il prossimo assegnamento.
;   L'algoritmo ritorna le sole DEFINIZIONI DI VARIABILI presenti in 'unvardefs'
;   coinvolte nel maggior numero di vincoli con le altre variabili definite in
;   'unvardefs'.
;   Per utilizzare correttamente questa funzione occorre specificare le definizioni
;   delle sole variabili non ancora assegnate.
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello di problema da analizzare. Necessario per determinare il numero di
;     vincoli cui ciascuna variabile
;   unvardefs (tipo: list)
;     La lista delle definizioni di variabili
; RITORNO:
;   (tipo: list)
;     La lista delle DEFINIZIONI DELLE VARIABILI aventi grado massimo (più vincolate).
; NOTE:
;   Il ritono è una lista: in caso di tie (pareggio) si dovrà filtrare ulteriormente.
(defun get-mcv-vardefs (csp unvardefs)
    (let ((max-deg (get-max (mapcar #'(lambda (x)
        (get-degrees csp unvardefs (car x))) unvardefs))))

        (remove-if
            #'(lambda (x) (< (get-degrees csp unvardefs (car x)) max-deg))
            unvardefs
        )
    )
)
; ------------------------------------------------------------------------------

; NOME:
;   get-degrees
; DESCRIZIONE:
;   Restituisce il numero di vincoli del problema 'csp' che legano la variabile 'var'
;   alle variabili definite in 'vardefs'.
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello del problema i cui vincoli devono essere analizzati.
;   vardefs (tipo: list)
;     La lista di DEFINIZIONI DELLE VARIABILI da analizzare.
```

```
;   var (tipo: symbol)
;     La variabile di cui misurare il grado.
; RITORNO:
;   (tipo: integer)
;     Il grado di 'var'.
(defun get-degrees (csp vardefs var)
    (length
        (loop for x in (csp-constraints csp)
            when (or (and (eql (cadr x) var) (assoc (caddr x) vardefs))
                     (and (eql (caddr x) var) (assoc (cadr x) vardefs)))
            collect x
        )
    )
)
; ---------------------------------------------------------------------------------

; NOME:
;   get-unassigned-variables
; DESCRIZIONE:
;   Restituisce la lista delle variabili del problema 'csp' non assegnate in base
;   all'ASSEGNAMENTO 'assignment'.
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello di problema (da cui ricavare l'insieme delle variabili).
;   assignment (tipo: list)
;     L'assegnamento da utilizzare per determinare le variabili non assegnate.
; RITORNO:
;   (tipo: list)
;     La lista dei nomi delle variabili non assegnate.
(defun get-unassigned-variables (csp assignment)
    (remove-if #'(lambda (x) (assoc x assignment)) (csp-variables csp))
)
; ---------------------------------------------------------------------------------

; NOME:
;   is-complete
; DESCRIZIONE:
;   Determina se l'assegnamento indicato risulta completo (ovvero coinvolge tutte le
;   variabili del problema 'csp').
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello di problema (da cui ricavare l'insieme delle variabili).
;   assignment (tipo: list)
;     L'assegnamento da analizzare.
; RITORNO:
;   (tipo: boolean)
;     Ritorna T se 'assignment' coinvolge tutte le variabili presenti in 'csp',
;     altrimenti ritorna NIL.
(defun is-complete (csp assignment)
    (endp (get-unassigned-variables csp assignment))
)
; ---------------------------------------------------------------------------------

; NOME:
;   get-variable-domain
; DESCRIZIONE:
;   Restituisce l'insieme dei valori del dominio della variabile 'var' nel problema
;   'csp'.
```

```
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello del problema da analizzare.
;   var (tipo: symbol)
;     La variabile di cui ricavare il dominio.
; RITORNO:
;   (tipo: list)
;     L'insieme dei valori del dominio della variabile 'var'.
(defun get-variable-domain (csp var)
    (cadr (assoc var (csp-domains csp)))
)
; ------------------------------------------------------------------------------

; NOME:
;   read-domains
; DESCRIZIONE:
;   Legge il FILE DEI DOMINII e ritorna la LISTA DEI DOMINII delle variabili definite.
; PARAMETRI:
;   file-path (tipo: string)
;     Il percorso del FILE DEI DOMINII.
; RITORNO:
;   (tipo: list)
;     La LISTA DEI DOMINII costruita.
(defun read-domains (file-path)
    (with-open-file (dom-file file-path :direction :input)
        (do ((ret-list nil)
             (curr-line (read dom-file nil) (read dom-file nil))
             (nrow 1 (1+ nrow)))
            ((endp curr-line) ret-list)

            (let ((var (car curr-line))
                  (domain (remove-duplicates (remove-if-not
                    #'integerp (cdr curr-line)))))
                (if (endp domain)
                    (error ERR-01 nrow var)
                    (enqueue ret-list (list (list var domain)))
                )
            )
        )
    )
)
; ------------------------------------------------------------------------------

; NOME:
;   read-constraints
; DESCRIZIONE:
;   Legge il FILE DEI VINCOLI e ritorna la LISTA DEI VINCOLI.
; PARAMETRI:
;   file-path (tipo: string)
;     Il percorso del FILE DEI VINCOLI.
;   csp-vars (tipo: list)
;     La lista delle variabili definite nel problema.
; RITORNO:
;   (tipo: list)
;     La LISTA DEI VINCOLI costruita.
(defun read-constraints (file-path csp-vars)
    (with-open-file (con-file file-path :direction :input)
        (do ((ret-list nil)
```

```
            (curr-line (read con-file nil) (read con-file nil))
            (nrow 1 (1+ nrow)))
           ((endp curr-line) ret-list)

           (let ((var1 (car curr-line))
                 (var2 (caddr curr-line))
                 (relop (cadr curr-line)))
               ; Controlla che cadr sia un rel-op.
               (unless (member relop '(= /= < <= > >=))
                   (error ERR-02 nrow relop)
               )
               ; Controlla che car e caddr siano variabili definite nel problema.
               (unless (and (symbolp var1) (member var1 csp-vars))
                   (error ERR-03 nrow var1)
               )
               (unless (and (symbolp var2) (member var2 csp-vars))
                   (error ERR-03 nrow var2)
               )
               (when (eq var1 var2)
                   (error ERR-04 nrow var1)
               )
               (enqueue ret-list (list (list relop var1 var2)))
           )
       )
   )
)
; -----------------------------------------------------------------------------

; NOME:
;   remove-variable-value
; DESCRIZIONE:
;   Rimuove il valore 'val' dal dominio della variabile 'var' nel problema 'csp'.
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello da modificare.
;   var (tipo: symbol)
;     La variabile il cui dominio deve essere modificato.
;   val (tipo: integer)
;     Il valore da rimuovere dal dominio.
; RITORNO:
;   (tipo: lista)
;     Il nuovo dominio (insieme di valori) associato a 'var'.
; SIDE-EFFECTS:
;   Modifica direttamente il modello 'csp'.
(defun remove-variable-value (csp var val)
    (setf (cadr (assoc var (csp-domains csp)))
        (remove val (cadr (assoc var (csp-domains csp))))
    )
)
; -----------------------------------------------------------------------------

; NOME:
;   set-variable-domain
; DESCRIZIONE:
;   Assegna il dominio (lista di valori) della variabile 'var' nel problema 'csp'.
; PARAMETRI:
;   csp (tipo: csp)
;     Il modello da modificare.
```

```
;   var (tipo: symbol)
;     La variabile il cui dominio deve essere impostato.
;   domain (tipo: list)
;     Il nuovo dominio da assegnare.
; RITORNO:
;   (tipo: list)
;     La nuova DEFINIZIONE DELLA VARIABILE 'var'.
; SIDE-EFFECTS:
;   Modifica direttamente il modello 'csp'.
(defun set-variable-domain (csp var domain)
    (rplacd (assoc var (csp-domains csp)) (list domain))
)
; ---------------------------------------------------------------------------
```